An Adaptive Algorithm for Efficient Message Diffusion in Unreliable Environments

Benoît Garbinato[⋆] Fernando Pedone[†] Rodrigo Schmidt[†]

[⋆]Université de Lausanne, CH-1015 Lausanne, Switzerland
Phone: +41 21 692 3409 Fax: +41 21 692 3405
E-mail: `benoit.garbinato@unil.ch`

[†]École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
Phone: +41 21 693 4797 Fax: +41 21 693 6600
E-mail: {`fernando.pedone, rodrigo.schmidt`}`@epfl.ch`

EPFL Technical Report IC/2004/30

19 March 2004

Abstract

In this paper, we propose a novel approach for solving the reliable broadcast problem in a probabilistic model, i.e., where links lose messages and where processes crash and recover probabilistically. Our approach consists in first defining the optimality of probabilistic reliable broadcast algorithms and the adaptiveness of algorithms that aim at converging toward such optimality. Then, we propose an algorithm that precisely converges toward the optimal behavior, thanks to an adaptive strategy based on Bayesian statistical inference. Our adaptive algorithm is modular and consists of two activities. The first activity is responsible for solving the reliable broadcast, given information about the failure probability of each link and of each process. This activity relies on the notion of Maximum Reliability Tree, which we derive from the notion of Maximum Spanning Tree. The other activity is responsible for approximating failure probabilities of links and processes, using Bayesian networks. We compare the performance of our algorithm with that of a typical gossip algorithm through simulation. Our results show, for example, that our adaptive algorithm quickly converges toward such exact knowledge.

Keywords: adaptive protocols, large-scale systems, probabilistic protocols, reliable broadcast, optimal message diffusion

1 Introduction

Diffusing information efficiently and reliably in an environment composed of many unreliable nodes interconnected by lossy communication links is an ability sought by many current large-scale systems (e.g., large-scale publish-subscribe architectures). Achieving reliable and efficient information diffusion in such contexts, however, is a complex task. Several reasons account for this fact. First, being composed of many components, it is unrealistic to assume that nodes have precise *a priori* information about the system characteristics, such as network topology and link reliability. Second, even if such information would be available to nodes at the beginning of the execution, the dynamic nature of a large system would render it obsolete quickly. Nodes, for example, may often leave the system, due to failures or explicit disconnections, constantly changing its topology. Finally, as observed by many researchers, mechanisms traditionally used to reliably broadcast information in small- and middle-size networks do not scale well when the system grows [2].

Many works have investigated this problem from a probabilistic perspective (e.g., [2, 4, 8, 9, 10, 11]). Probabilistic algorithms scale much better than deterministic ones and achieve high reliability. Intuitively, every node that receives a message chooses a subset of system members, for example among the complete set of destinations, and propagates (i.e., gossips) the message to these nodes. The gossip nature of the algorithm combined with the possibility of crashes and message loss implies that there are some chances that not all nodes receive the original message. Nevertheless, provided that nodes keep gossiping the original message "long enough" it can be guaranteed that with very high probability all nodes receive the message.

In this paper, we propose an approach to improve the performance of gossip-based algorithms by taking into account the topology and probabilistic nature (i.e., node failure and message loss probabilities) of the environment in which these algorithms execute. Since nodes adapt to the environment characteristics during the execution, we call such algorithms *adaptive*. This adaptive characteristic is precisely what distinguishes our approach from previous works, which in general do not take topology and reliability aspects into account to improve performance. As we discuss in the paper, our approach is complementary to previous optimizations proposed in the literature (e.g., [11]) and could be combined with them.

The motivation for adaptive algorithms is performance. Large-scale systems are usually composed of several parts with varying reliability characteristics (e.g., local-area network links are usually more reliable than wide-area network links), and adjusting the gossip mechanism according to the system characteristics can provide more efficient results. To better spell out our argument, consider the following simple example in which two nodes are connected through two independent paths. Path one loses messages with probability L, $0 < L < 1$. Path two is less reliable than path one and loses messages with probability αL, where $\alpha > 1$. With a typical gossip algorithm, which chooses paths randomly for every send, after node one sends k_0 messages to node two, the probability that at least one message reaches node two is $1-(\sqrt{\alpha}\, L)^{k_0}$ (see Appendix A). Using an algorithm adapted to this environment, which chooses the paths according to their reliability probabilities (and therefore always chooses the first path), node one reaches node two with probability $1 - L^{k_1}$ after k_1 messages are sent.

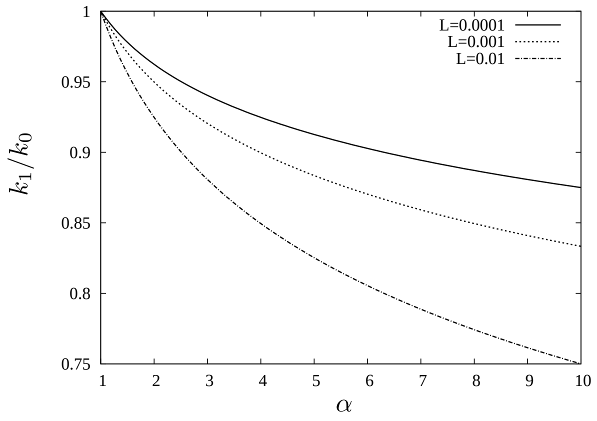


Figure 1: Adaptive versus traditional gossip

Consequently, to reach the same reliability as an environment-adapted algorithm, a typical gossip algorithm has to retransmit more messages, wasting throughput and unnecessarily consuming system resources. Figure 1 depicts the relation between k_0 and k_1 as a function of α when both algorithms achieve the same reliability. When $\alpha = 1$, both paths have the same reliability and so, there is no difference between a typical gossip algorithm and an environment-adapted one. When $\alpha = 10$, even if path one is very reliable, for example $L = 0.0001$, an adaptive algorithm only needs about 87% of the messages sent by a traditional gossip algorithm to reach the same overall reliability. This means that even in a very simple configuration, a traditional gossip algorithm would waste about 13% of throughput only retransmitting messages. When paths are less reliable (e.g., $L = 0.01$ in Figure 1) and in more complex topologies, further improvements are obtained. Section 5 discusses this issue in details, using a more sophisticated traditional gossip algorithm.

Briefly, in our approach each time a node decides to broadcast a message, it builds a *Maximal Reliability Tree (MRT)*, a spanning tree that determines the best way to propagate messages. To build an MRT, nodes use information about the system topology and the reliability of nodes and communication links. The more precise this information, the closer to the optimal the gossiping mechanism will be. An optimal algorithm, for example, will try to avoid gossiping messages through very unreliable links, if an alternative more reliable path exists. We initially assume that broadcasting nodes have perfectly accurate information about the system topology and the nodes and links reliability to build the MRT at a given time. Although this leads to an optimal reliable broadcast algorithm, it is mostly of theoretical interest only, since such complete information is difficult to obtain. Then, we replace the full-knowledge assumption with a more realistic one in which nodes try to approximate the topology and the reliability parameters of the system during the execution, adapting to changes. This results in a modular and simple design. Our optimal reliable broadcast algorithm (based on perfect knowledge about the system) remains the same, while our adaptive strategy is completely encapsulated in a separate activity that precisely tries to approximate such perfect knowledge. We believe that this approach could be used to develop other adaptive algorithms in large-scale environments.

Intuitively, our approximation strategy works as follows. First, nodes keep exchanging their local knowledge of the network topology with their direct neighbors. This guarantees that each node will eventually discover the complete network topology. Second, nodes monitor their direct neighbors and try to assess their availability and the reliability of the communication links interconnecting them. This information is also part of the messages exchanged between neighbors and eventually reaches all nodes connected to the system. Upon receiving a message from a neighbor, a node updates its local information. This process uses a mix of Bayesian networks and a distortion factor. The distortion factor tries to take into account the information aging via two factors, namely time and distance. Each approximated data has a distortion factor that is proportional to how much time ran out since it was last updated, and to how far in the network it was originated. We show that provided that the systems' characteristics remain stable for some time, the topology and reliability information assessed by the nodes eventually converge toward a perfect knowledge of the system. Then, the performance of our adaptive reliable broadcast algorithm coincides with the performance of the optimal reliable broadcast algorithm. Finally, although nodes keep exchanging information with their neighbors, this data can also be opportunistically piggybacked in gossip messages, saving communication bandwidth.

The rest of the paper is organized as follows. Section 2 introduces the system model and the concepts of optimal and adaptive reliable broadcast algorithms. Section 3 describes an optimal algorithm to solve probabilistic reliable broadcast. Section 4 presents our adaptive algorithm, which is derived from the optimal one. Section 5 evaluates our approach through simulation. Section 6 reviews related work, and Section 7 concludes the paper.

2 Probabilistic Model and Definitions

2.1 Processes and Communication Links

We consider a system of distributed processes communicating by message passing. There are no strong assumptions about the time it takes for processes to execute and for messages to be transmitted. The system's topology is defined by $G = (\Pi, \Lambda)$, where $\Pi = \{p_1, p_2, ...\}$ is a set of processes ($|\Pi| = n$), and $\Lambda = \{l_1, l_2, ...\} \subseteq \Pi \times \Pi$ is a set of *bidirectional* communication links. A link l_x from p_i to p_j is also denoted by $l_{i,j}$. If $l_{i,j} \in \Lambda$ and $i \neq j$, we say that p_j is neighbor of p_i. The set of all p_i's neighbors is denoted by $neighbors(p_i)$. We define a *path* as a combination of links and intermediate processes through which a message can transit to reach a destination.

Processes can crash and subsequently recover and links can lose messages. We do not consider Byzantine failures, i.e., processes execute according to their algorithms. Processes have access to local *volatile memory* and *stable storage.* Information recorded in stable storage survives crashes, which is not the case for information stored in volatile memory. Processes should be judicious about using stable storage, however, since it is significantly slower than volatile memory.

Processes execute a sequence of steps, which can be of two kinds. In a *normal* step, a process (a) may receive a message from one of its neighbors or send a message to one of its neighbors (but not both), (b) undergo a state transition, and (c) may write some information in stable storage. These assumptions simplify the probabilistic analysis and proofs of our algorithms. In a *crashed* step, the process simply loses all the contents of its volatile memory, if any, and passes to the next step, which may be normal or crashed. If p_i executes a crashed step s_k followed by a normal step s_{k+1}, we say that p_i has *recovered* at step s_{k+1}.

A *configuration* $C = (P_1, P_2, ..., P_{|\Pi|}, L_1, L_2, ..., L_{|\Lambda|})$ is a tuple of probabilities, where P_i is the ratio between the number of crashed steps and the total number of steps executed by p_i in some execution of the algorithm, and L_x is the ratio between the number of messages lost by l_x and the total number of messages transmitted through l_x in the execution. P_i can be understood as the probability that process p_i executes a crashed step in the execution and L_x as the probability that link l_x loses a message, whenever it is requested to transmit one.

2.2 Probabilistic Reliable Broadcast

Reliable broadcast is defined by the primitives broadcast(m) and deliver(m). To simplify the discussion, we assume that processes in Π are part of a single broadcast group; in practice, there may exist several broadcast groups, with processes possibly being members of more than one group. A probabilistic reliable broadcast algorithm $\mathcal{A}^K$ ensures with at least probability K that if a process in Π delivers some message m, then all processes in Π will deliver m. For brevity, we do not require a message to be delivered exactly once by each process. Usually, to ensure exactly-once message delivery in a crash/recovery model, processes have to do some local logging to keep track of messages already delivered. If desired, such a guarantee can be built on top of our reliable broadcast primitive.

2.3 Adaptation and Optimality

To compare the efficiency of different probabilistic reliable broadcast algorithms, we consider the number of messages exchanged. According to this parameter, it seems intuitive that processes should privilege paths requiring the lowest possible number of retransmissions to reach other processes. Our definition of adaptation is based on the notion of optimal algorithms. We informally define optimal and adaptive probabilistic reliable broadcast algorithm as follows.

Definition 1 *A probabilistic reliable broadcast algorithm $\mathcal{O}^K$ is optimal to some configuration C w.r.t. the number of messages if there is no algorithm $\mathcal{X}^K$ such that processes executing $\mathcal{X}^K$ in C exchange fewer messages than processes executing $\mathcal{O}^K$ in C.*

Definition 2 *A probabilistic reliable broadcast algorithm $\mathcal{A}^K$ is adaptive to some configuration C iff the number of messages exchanged by processes executing $\mathcal{A}^K$ in C in response to a broadcast is eventually equal to the number of messages exchanged by processes executing $\mathcal{O}^K$ in C.*

3 An Optimal Algorithm

The optimal algorithm we propose relies on the assumption that each process knows the topology G and the failure configuration C (i.e., crash and loss probabilities) of the system. Each process then uses this knowledge to minimize the number of messages needed to reach all processes with a given probability. This is achieved by having each process first compute a *Maximum Reliability Tree* (MRT) of the system, as described next.

3.1 Maximum Reliability Tree (MRT)

The Maximum Reliability Tree is a spanning tree containing the most reliable paths in G connecting all processes in Π. We assume that the MRT is calculated by function $mrt(G, C)$ using a modified version of Prim's algorithm [1] (see Appendices B and C). If processes agree on the system's topology and configuration, they all build the same MRT. Under more realistic assumptions, however, processes may have different views of the system topology and configuration. In such cases, they will build different MRT's. To avoid ambiguity, we denote $mrt_i(G, C)$ the MRT built by some process p_i. Notice that since MRT is a tree, it always contains exactly $n-1$ links.

3.2 From MRT to Optimal Algorithm

Intuitively, given a sender p_i, our optimal algorithm uses $mrt_i(G, C)$ to determine the minimum necessary number of messages that must transit through each edge in order to reach all processes with probability K. To state this idea more formally, we label p_s the root of the tree and p_i all other processes in $mrt_s(G, C)$, with $1 \leq i < |\Pi| - 1$. Then, we label l_i the link that leads to p_i and m_i the number of messages going through l_i.[1] Figure 2 illustrates this labeling on a concrete example. So, we can now restate the intuitive idea of our optimal algorithm as follows: it consists in minimizing the sum of m_i necessary to reach all processes with some probability K.

In order to present our algorithm, we must still introduce a few additional notions and terms. First, we define T_i to be the *subtree of* $mrt_s(G, C)$ *with* p_i *as root*; from this definition, we have that $T_s = mrt_s(G, C)$. Then, we define S_i to be the set of *direct subtrees of* p_i, i.e., S_i contains any subtree whose root is a process p_j directly connected to p_i via link l_j. Figure 3 illustrates this notion of direct subtrees on the maximum reliability tree introduced in Figure 2

[1] With this labeling, we can simplify the way we write the loss probability of each link l_i as L_i.

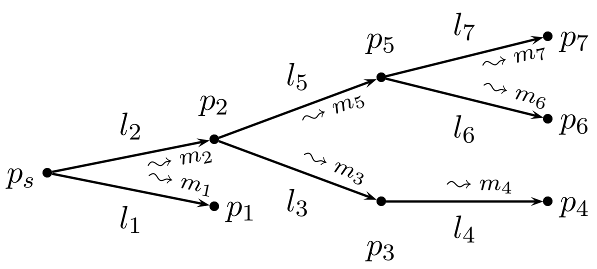


Figure 2: A relabeled MRT

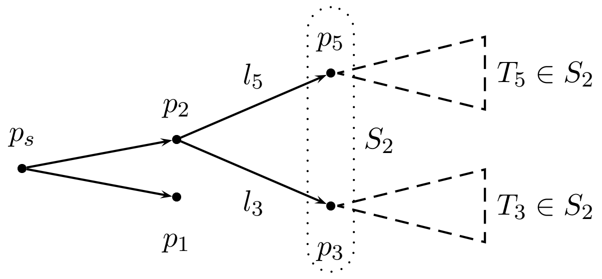


Figure 3: Direct subtrees in mrt_s

(e.g., $S_2 = \{T_3, T_5\}$). Finally, we define $\vec{m}_i$ to be a vector whose components are the numbers of messages transiting through the links of T_i.

Given this terminology, we can now introduce the *reach* function: given a tree T_i and a vector $\vec{m}_i$, this function computes the probability that all processes in T_i are reached by at least one message. Eq. (1) presents the *reach* function in a recursive form with $\vec{m}_i[j]$ being the *j-th* component of vector $\vec{m}_i$. The idea consists in multiplying the probability that at least one message reaches the root process p_j of each subtree $T_j \in S_i$ by the recursive probability to reach all processes of T_j. Then, if process p_j is a leaf ($T_j = \perp$), we have that $|\vec{m}_j| = 0$ and $reach(\perp, \vec{0}) = 1$.

$$reach(T_i, \vec{m}_i) = \begin{cases} 1 & \text{if } T_i = \perp \\ \prod_{T_j \in S_i} (1 - [1 - (1 - P_i) \times (1 - L_j) \times (1 - P_j)]^{\vec{m}_i[j]}) \times reach(T_j, \vec{m}_j) & \text{otherwise} \end{cases} \quad (1)$$

Since Eq. (1) presents a typical tail-recursion form, we can also write the *reach* function in pure iterative form, as shown by Eq. (2), with $pred(j)$ being the process that precedes p_j in T_i.

$$reach(T_i, \vec{m}_i) = \prod_{j=1}^{n-1} 1 - [1 - (1 - P_{pred(j)}) \times (1 - L_j) \times (1 - P_j)]^{\vec{m}_i[j]} \quad (2)$$

Using the *reach* function, we can state our optimization problem in a concise manner, as shown in Eq. (3), where λ_j expresses $1 - (1 - P_{pred(j)}) \times (1 - L_j) \times (1 - P_j)$.

$$\begin{array}{llcl} minimize & c(\vec{m}) & = & \sum_{j=1}^{|\vec{m}|} m[j] \\ subject\ to & r(\vec{m}) & = & \prod_{j=1}^{|\vec{m}|} 1 - \lambda_j^{m[j]} \quad \geq \quad K \end{array} \quad (3)$$

We encapsulate the solution to this optimization problem in the *optimize*() function, which takes an MRT and K as input parameters and returns a vector $\vec{m}_s$. Algorithm 1 shows how the optimize function is used to implement our optimal probabilistic reliable broadcast.

Algorithm 1 Optimal Algorithm at p_k

```
 1: To execute broadcast($m$) do
 2:    $mrt_k \leftarrow mrt_k(G, C)$
 3:    $propagate(m, mrt_k, p_k)$
 4:    deliver($m$)

 5: when receive ($m$, $mrt_j$) for the first time
 6:    $propagate(m, mrt_j, p_k)$
 7:    deliver($m$)

 8: function $propagate(m, mrt_j, p_k)$
 9:    $\vec{m}_j \leftarrow optimize(mrt_j, K)$
10:    for all subtree $T_i \in S_{j,k}$ do
11:       repeat $\vec{m}_j[i]$ times
12:          send ($m$, $mrt_j$) to $p_i$
```

3.3 The *optimize*() Function

Algorithm 2 implements *optimize*() via a greedy strategy. Our algorithm works by optimizing each individual step, hoping that the resulting global solution will be optimal. From operational research it follows that a greedy algorithm does indeed yield an optimal solution if the problem it solves is itself greedy (a fact we prove in Appendix D). The algorithm starts with a minimal solution, i.e., an initial vector $\vec{m}$ of the form $(1, 1, ..., 1)$, and then proceeds in incremental steps. In each step, the algorithm chooses the link l_j in the MRT that maximizes the gain in terms of the probability to reach all processes when sending one more message through l_j. It then stops when the desired probability K is reached and returns vector $\vec{m}$ as solution. In Algorithm 2, $\vec{u}_j$ denotes a vector in which the j-th element is 1 and the others are 0, e.g., $\vec{u}_2 = (0, 1, 0, ..., 0)$.

Algorithm 2 A Greedy Algorithm for *optimize*()

```
1: function $optimize(mrt, K)$
2:    $\vec{m} \leftarrow (1, 1, 1, \cdots, 1)$
3:    while $r(\vec{m}) < K$ do
4:       let $\vec{u}_j$ be such that $\frac{r(\vec{m}+\vec{u}_j)}{r(\vec{m})}$ is maximum
5:       $\vec{m} \leftarrow \vec{m} + \vec{u}_j$
6:    return $\vec{m}$
```

4 An Adaptive Algorithm

4.1 Overview of the Algorithm

Our adaptive protocol is based on Algorithm 1, used by the optimal protocol. The difference lies in the knowledge processes have about the topology $G = (\Pi, \Lambda)$ and the configuration C. In the optimal protocol, this knowledge is accurate; in the adaptive protocol, it is an approximation. Thus, with the adaptive protocol, in addition to executing Algorithm 1, processes are constantly trying to approximate G and C based on what they observe from the system. If G and C remain stable for "long enough", our adaptive protocol converges toward the optimal one.

Network topology (G). Initially, processes know only the links connecting them directly to their neighbors—notice that we do not require processes to agree on the system membership at any given time. To share this knowledge, each process periodically sends heartbeat messages containing its view of the topology to all its neighbors. When receiving a heartbeat, a process updates its topology knowledge with the information received. The next time this process propagates its topology view, it will include the recently added information. If the network topology remains stable and partitions are temporary, even in the presence of process crashes and message losses processes eventually learn the global system topology.

Reliability configuration (C). Heartbeats are also used by processes to determine the reliability of the system and to share this information with other processes. The probability of crashing is approximated by the process itself by periodically reading the value of its local clock and storing it in stable storage. When the process recovers from a crash, it reads the last clock value from stable storage and compares it to the current time. The probability of failure is proportional to the number of intervals missed during some sufficiently large amount of time. When a process p_k receives a heartbeat from some neighbor p_j, it updates its local estimate of p_j's failure probability by simply adopting the value received from p_j. In addition, p_k adjusts the message loss probability of link $l_{k,j}$. If p_k does not receive any heartbeats from p_j for some time, p_k increases the failure probability of p_j and the message loss probability of $l_{k,j}$. To approximate the reliability of non-neighbor processes and remote links, p_k only relies on information received from its neighbors. When p_k receives a heartbeat with C_j from its neighbor p_j, it must decide which estimates to keep, i.e., its current ones or the ones in C_j. Intuitively, the idea is to choose the *less distorted* estimates. This implies that each estimate has a distortion factor, which expresses how accurate the estimate is: the higher the factor, the less accurate the estimate. As explained in next section, two factors tend to erode an estimate accuracy: *time* and *distance*.

4.2 A Detailed Approximation Algorithm

Algorithm 4 presents our solution to approximate the knowledge some process p_k has about G and C. To simplify the algorithm, we assume that p_k knows Π, the set of processes in the system, right from the start—this assumption is not essential and can be removed at the cost of some additional complexity in the algorithm.[2] Thus p_k must approximate Λ and C. In Algorithm 4, Λ_k and C_k denote the view p_k has on Λ and C, respectively, at any given time.

Data structures. The two main data structures of Algorithm 4 are Λ_k and C_k. While Λ_k has exactly the same structure as Λ (i.e., a set of links), C_k is more complex than C. Hereafter $C_k[p_i]$ denotes P_i, the crash probability of p_i at p_k, and $C_k[l_j]$ denotes L_j, the message loss probability of l_j at p_k. $C_k[p_i]$ and $C_k[l_j]$ are complex data structures representing p_k's current estimates of P_i and L_j, respectively—we refer to such data structures as simply *estimates*. An estimate contains a small Bayesian network used to approximate P_i and L_j. Section 4.3 describes how Bayesian networks are used to compute such probabilities via functions *initializeReliability*(), *increaseReliability*() and *decreaseReliability*(). In addition to Bayesian networks, estimates contain several other fields, listed and initialized between Lines 2 and 12, and explained next.

Algorithm structure. Algorithm 4 is an epidemic-type protocol: each process p_k periodically sends its Λ_k and C_k approximation to its neighbors; the periodicity is set to δ and also serves as a heartbeat protocol to detect process crashes and messages losses. This epidemic-type propagation is shown on Lines 14 to 17. Although these messages are completely independent of the application, the information they convey could be piggybacked into application messages.

Approximating Λ. Process p_k initializes Λ_k with the links to its neighbors (Line 9). Whenever p_k receives (Λ_j, C_j) from some neighbor p_j, it adds all links in Λ_j to Λ_k (Line 33.) Next time p_k sends its view of Λ_k to its neighbors, Λ_k will contain these additional links. As already discussed, this strategy ensures that Λ_k will eventually embrace the complete topology, i.e., it will eventually converge to Λ.

[2]Additional complexity here means using dynamic data structures instead of static ones.

Approximating C. To approximate C (i.e., the crash probability of processes in Π and the message loss probability of links in Λ), p_k relies on the four events presented next.

- **Event 1.** *Reception of* (Λ_j, C_j) *from neighbor* p_j (Lines 18–33). This event allows p_k to know how many messages were lost by link $l_{k,j}$. Each heartbeat sent by p_j holds a sequence number in $C_j[p_j].seq$. Similarly, p_k keeps in $C_k[p_j].seq$ the sequence number of the last heartbeat received from p_j and in $C_k[p_j].suspected$ the number of times it suspected $l_{k,j}$ since the last time it received a heartbeat from p_j. Based on this information, p_k can proportionally adjust the message loss probability of $l_{k,j}$ (Line 19) and decide whether the suspicion timeout associated with p_j should be adjusted (Line 23).

  Process p_k also uses C_j to select and adopt the best estimate for each process and for each link. This is done by function *selectBestEstimate*() presented in Algorithm 3. This function selects the best estimate based on the notion of distortion factor. Intuitively, for any p_i, the corresponding distortion factor $C_k[p_i].d$ is proportional both to the network distance between p_k and p_i, and to how much time ran out since p_k last updated its estimate about p_i. A similar principle applies to the estimate of any link l_i, except that in this case the distortion factor merely captures the distance between p_k and l_i. The minimal value of $C_k[p_i].d$ is given by the network distance between p_k and p_i, and $C_k[p_i].d$ increases as p_k hears nothing about p_i (directly or indirectly) for a given period of time (timeout $\Delta_k[p_i]$). This is why process estimates in C_k have their distortion factor initialized to ∞: initially, p_k knows nothing about the failure probabilities of other processes. For its own probability and the probability of direct links, the distortion factor is 0.

  Given two distortion factors, selecting the best estimate means adopting the less distorted one. In addition, when adopting p_j's estimate, process p_k also increments the corresponding distortion factor. This accounts for the fact that the estimate p_k just adopted is now second-hand. Note that having the distortion factor $C_j[p_j].d = 0$ guarantees that the estimate of p_j concerning its own reliability will always be adopted by p_k. Finally, selecting the best estimates only makes sense for links that are already known to p_k. For new links, p_k merely adopts p_j's estimate and adjusts the distortion factor (Lines 30–32).

- **Event 2.** *No update of* p_j*'s estimate for* $\Delta_k[p_j]$ *time* (Lines 34–39). The distortion factor associated with some estimate $C_k[p_j]$ captures the fact that in absence of news about p_j, its estimate should get more distorted. This increase in distortion is captured by incrementing $C_k[p_j].d$ (Line 35). If p_j is also a neighbor of p_k, the absence of update means that p_k did not receive any heartbeats from p_j for some time, and so, it should suspect it. Furthermore, both p_j and the link to it should have their estimated reliability decreased (Lines 38–39).

- **Events 3 and 4.** *No crash of p_k during Δ_{up} time, and returning from a crash lasting Δ_{down} time* (Lines 40–43). The last two events help augment p_k's knowledge about its own reliability. The idea is to increase or decrease p_k's estimate of its own reliability *proportionally* to how long it stayed up and down.

Algorithm 3 Best estimate selection at process p_k

```
1: function selectBestEstimate($e_k, e_j$)
2:   if $e_j.d < e_k.d$ then                {less distorted is best}
3:     $e_k \leftarrow e_j$                 {adopt the best and}
4:     $e_k.d \leftarrow e_k.d + 1$         {adjust distortion}
```

4.3 Bayesian Networks

To estimate the failure probability of some process or link, p_k builds a list of probability intervals and maintains for each interval a *belief* that the failure probability lies within the corresponding interval. In doing so, p_k actually builds a small Bayesian network $b \rightarrow s$, where b is the belief and s is the failure probability. Functions *initializeReliability*(), *decreaseReliability*() and *increaseReliability*() are responsible for managing such Bayesian networks (see Algorithm 5).

Let F be the event associated with the crash of some process, the message loss of some link, or merely the *suspicion* that such a crash or loss occurred. We denote by $P_{F|B}[u]$ the u-th probability interval associated with F at p_k, and by $P_B[u]$ the corresponding belief, i.e., the probability that the "real" failure probability in C lies within the u-th interval. In Algorithm 5 we consider U failure probability intervals (Line 2), initially associated to identical beliefs (Lines 5 to 7).

To compute the new degree of belief B on a given interval u, based on the observation of an event F, p_k uses basic conditional probability $P_{B|F}[u] \times P_F = P_{F|B}[u] \times P_B[u]$ and Bayes theorem given by Eq. (4). This equation is used to compute the belief *a posteriori* on $P_{F|B}[u]$ (denoted by $P_{B|F}[u]$), which will be the new value of $P_B[u]$ after event F has been observed by p_k. This is precisely what function *decreaseReliability*() of Algorithm 5 does (Lines 8 to 11).

Algorithm 4 Approximating (Λ_k, C_k) at process p_k

```
1: INITIALIZATION:

2:    for all $p_i \in \Pi$ do                                  {initialize all process estimates with:}
3:       $initializeReliability(C_k[p_i])$                      {a set of reliability beliefs}
4:       $C_k[p_i].d \leftarrow \infty$                          {a distortion factor}
5:       $C_k[p_i].seq \leftarrow 0$                             {a heartbeat sequencer}
6:       $C_k[p_i].suspected \leftarrow 0$                       {a suspicion counter}
7:       $\Delta_k[p_i] \leftarrow \delta$                       {a heartbeat timeout}
8:    $C_k[p_k].d \leftarrow 0$                                  {$p_k$ sees itself with no distortion}

9:    $\Lambda_k \leftarrow \{l_{k,i} \mid p_i \in \text{neighbors}(p_k)\}$   {first $p_k$ only knows about neighbor links}
10:   for all $l_i \in \Lambda_k$ do                            {initialize these link estimates with:}
11:      $initializeReliability(C_k[l_i])$                      {a set of reliability beliefs}
12:      $C_k[l_i].d \leftarrow 0$                               {a distortion factor}

13: TO UPDATE $(\Lambda_k, C_k)$:

14:   every $\Delta_k[p_k]$ do :                                 {do periodically:}
15:      $C_k[p_k].seq \leftarrow C_k[p_k].seq + 1$              {increment heartbeat sequencer}
16:      for all $p_i \in neighbors(p_k)$ do                     {propagate my estimates via}
17:         send $(\Lambda_k, C_k)$ to $p_i$                     {a sequenced heartbeat}

18:   when received $(\Lambda_j, C_j)$ from $p_j$ do             {Event 1}
19:      $adjust \leftarrow C_k[p_j].suspected - (C_j[p_j].seq - C_k[p_j].seq)$   {adjustment factor for link suspicions}
20:      $C_k[p_j].suspected \leftarrow 0$                       {and cancel all suspicions}
21:      if $adjust > 0$ then                                   {link $l_{k,j}$ was suspected too much}
22:         $increaseReliability(C_k[l_{k,j}], adjust)$          {update $p_k$'s belief about that link}
23:         if $adjust > 1$ then $\Delta_k[p_j] \leftarrow \Delta_k[p_j] + \delta$   {so adjust its associated timeout}
24:      if $adjust < 0$ then                                   {link $l_{k,j}$ was not suspected enough}
25:         $decreaseReliability(C_k[l_{k,j}], |adjust|)$        {update $p_k$'s belief about that link}
26:      for all $p_i \in \Pi$ do                                {for each process, take}
27:         $selectBestEstimate(C_k[p_i], C_j[p_i])$             {the most accurate estimate}
28:      for all $l_i \in (\Lambda_k \cap \Lambda_j)$ do          {for each common link, take}
29:         $selectBestEstimate(C_k[l_i], C_j[l_i])$             {the most accurate estimate}
30:      for all $l_i \in \Lambda_j - (\Lambda_k \cap \Lambda_j)$ do   {for new links, simply}
31:         $C_k[l_i] \leftarrow C_j[l_i]$                       {adopt $p_j$'s estimate}
32:         $C_k[l_i].d \leftarrow C_k[l_i].d + 1$               {and adjust distortion}
33:      $\Lambda_k \leftarrow \Lambda_k \cup \Lambda_j$          {finally, merge topology knowledge}

34:   when not[updated $C_k[p_j]$, $p_j \neq p_k$, in the last $\Delta_k[p_j]$] do   {Event 2}
35:      $C_k[p_j].d \leftarrow C_k[p_j].d + 1$                  {knowledge gets distorted with time}
36:      if $p_j \in neighbors(p_k)$ then                        {in addition, neighbors should be}
37:         $C_k[p_j].suspected \leftarrow C_k[p_j].suspected + 1$   {suspected and their reliability}
38:         $decreaseReliability(C_k[p_j], 1)$                   {beliefs should be updated}
39:         $decreaseReliability(C_k[l_{k,j}], 1)$               {as well as the link to them}

40:   every $\Delta_{tick}$ do                                   {Event 3}
41:      $increaseReliability(C_k[p_k], 1)$                      {update $p_k$'s belief about itself}

42:   when recovering from a crash lasting $n \times \Delta_{tick}$ do   {Event 4}
43:      $decreaseReliability(C_k[p_k], n)$                      {update $p_k$'s belief about itself}
```

Algorithm 5 Reliability beliefs management

```
1: Initialization
2:   $U \leftarrow 100$        {precision of probabilistic intervals}
3: function initializeReliability(estimate)
4:   with estimate do
5:     for all $u = 1..U$ do
6:       $P_{F|B}[u] \leftarrow \frac{2u-1}{2U}$        {probabilistic intervals}
7:       $P_B[u] \leftarrow \frac{1}{U}$        {with equal initial beliefs}
8: function decreaseReliability(estimate, factor)
9:   with estimate repeat factor times
10:    for all $u = 1..U$ do
11:      $P_B[u] \leftarrow \frac{P_B[u] \times P_{F|B}[u]}{\sum_{v=1}^{U} P_B[v] \times P_{F|B}[v]}$
12: function increaseReliability(estimate, factor)
13:   with estimate repeat factor times
14:     for all $u = 1..U$ do
15:       $P_B[u] \leftarrow \frac{P_B[u] \times (1-P_{F|B}[u])}{\sum_{v=1}^{U} P_B[v] \times (1-P_{F|B}[v])}$
```

As shown in function *increaseReliability*(), a similar computation is performed to account for the absence of failure (Lines 12 to 15).

$$P_{B|F}[u] = \frac{P_{F|B}[u] \times P_B[u]}{\sum_{v=1}^{U} P_{F|B}[v] \times P_B[v]} \quad (4)$$

Table 1 illustrates how Bayesian approximation works. The example starts with an initial configuration with equal *a priori* beliefs for $U = 5$ (case a). Then, it shows how the beliefs have been adapted after a suspicion (case b). Since the real probability must fall into some probability interval of Table 1, we have that $\sum_u C_k[p_i].P_B[u] = 1$ is an invariant of Algorithm 4.

u	$C_k[p_i].P_{F|B}[u]$	$C_k[p_i].P_B[u]$
1	[0.0 , 0.2)	0.2
2	[0.2 , 0.4)	0.2
3	[0.4 , 0.6)	0.2
4	[0.6 , 0.8)	0.2
5	[0.8 , 1.0]	0.2

(a) Initial configuration

u	$C_k[p_i].P_{F|B}[u]$	$C_k[p_i].P_B[u]$
1	[0.0 , 0.2)	0.04
2	[0.2 , 0.4)	0.12
3	[0.4 , 0.6)	0.20
4	[0.6 , 0.8)	0.28
5	[0.8 , 1.0]	0.36

(b) After a failure suspicion

Table 1: Adapting failure beliefs after a suspicion

5 Simulation Results

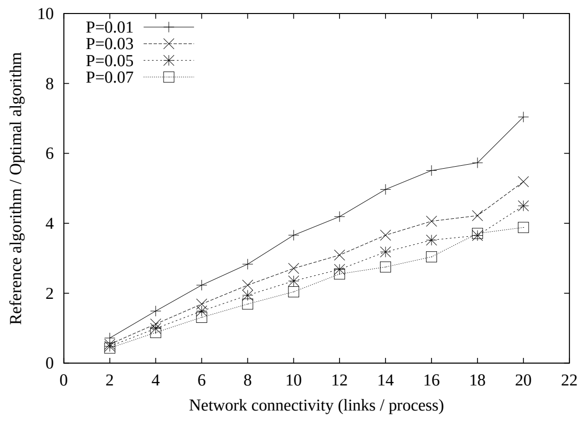
In order to evaluate the performance of our adaptive algorithm we built a discrete-event simulation model and conducted several experiments with it. Our model simulates the behavior of processes and links in a distributed system, associating a crash probability to each process and a loss probability to each link. To simplify the interpretation of our results, we considered that all processes have the same crash probability P and that all links have the same loss probability L. This choice counts against our adaptive algorithm because contrary to traditional gossip, our solution selects the most reliable links. Nevertheless, even under such unfavorable conditions, the results provide strong evidence about the benefits of an adaptive strategy.

We performed experiments with 100 processes for several network topologies. In the minimal network connectivity setup each process had two neighbors (i.e., the network is a ring). The connectivity was increased until each process had 20 neighbors. Heartbeat messages were 50K bytes long and contained a small Bayesian network per process, information about the loss probability of links, and some additional fields as described in Section 4.2.

Our results were compared to a reference algorithm, implementing a typical gossip-based reliable broadcast. The execution proceeds in steps, and in each step processes forward data messages to their neighbors. The execution continues until all processes have been reached with probability 0.9999—the exact number of steps needed depends on the parameters of a particular setup and were determined interactively. As a simple optimization, processes acknowledge the receipt of data messages. Thus, when choosing the neighbors to which some data message m will be forwarded, each process p never forwards m to its neighbor q if (a) it has previously received m from q, or (b) it has received an acknowledgment message from q for m.

In Figure 4 we compare the adaptive and the reference algorithms. In Figure 4(a), we varied the crash probability while assuming that links were reliable (i.e., $L = 0$); in Figure 4(b) we varied the message loss probability while assuming that processes were reliable (i.e., $P = 0$). In both figures, the y-axis shows the ratio between the number of messages sent by the reference algorithm and by the adaptive algorithm to reach all processes with the same probability. For example, when the connectivity is 16 and $P = 0.03$, the adaptive algorithm needs 4 times fewer messages than the reference algorithm to reach all processes with the same probability. The adaptive algorithm provides better results as the connectivity of the network increases. This is due to the fact that in low-connected graphs in which processes and link have the same reliability, the adaptive algorithm does not have much room for improving the forwarding mechanism.

Figure 5 shows the effort needed to converge (i.e., all processes in the system learn the reliability probabilities) in number of messages per link. This parameter is twice the number of heartbeat messages sent by a process through a link until all processes converge. For example, when the network connectivity is 6 and $L = 0.05$, about 400 heartbeat messages will be sent


(a)

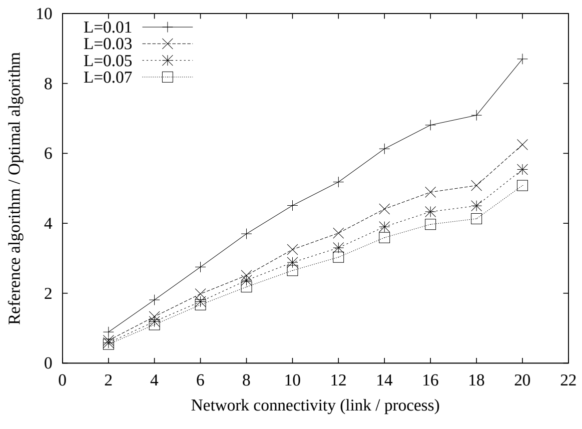
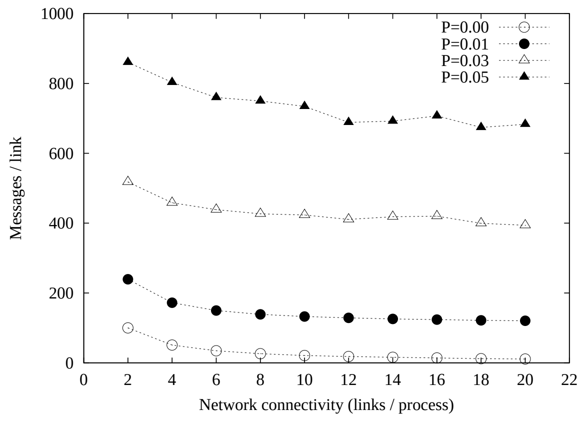

(b)

Figure 4: Algorithms with (a) reliable links and (b) reliable processes

per process through a link. If heartbeats are sent each 1 second, the adaptive mechanism will converge in about 7 minutes. Two factors amount for the convergence time: the time it takes for the Bayesian networks to find the right probability interval accurately—in the simulations we used 100 probability intervals—and the time it takes for this information to reach all processes.


(a)

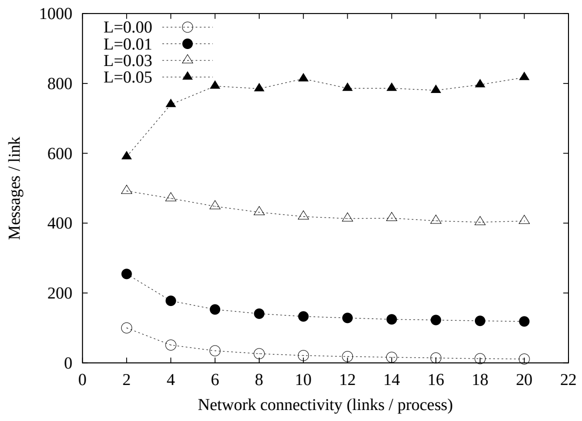

(b)

Figure 5: Convergence with (a) reliable links and (b) reliable processes

Connectivity has a double effect on convergence. On the one hand it helps convergence since it reduces the time it takes for the inferred information to arrive at all processes. On the other hand, it hurts convergence since as more links are added, more information has to be inferred. We have also observed that low probabilities are easier to be inferred by our Bayesian model than high probabilities. In the case of links, the effects are more noticeable since links are more numerous than processes. This can be observed in Figure 5(b) when $L = 0.05$.

To evaluate the scalability of our adaptive algorithm, we executed simulations using two types of network topologies: a ring (i.e., each process connected to two others) and a random tree. In both cases about 100 graphs were generated for each experiment (see Figure 6). The ring is a worst-case topology in which messages should traverse in the average half the processes in the network. In such a case, the convergence time increases linearly with the size of the system. For random trees, however, the convergence time is almost constant . In practical scenarios, the topology is expected to be closer to a tree than to a ring.

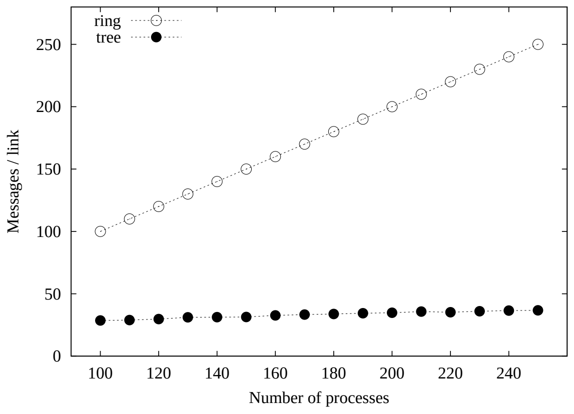


Figure 6: Algorithm scalability

6 Related Work

Epidemic protocols, also known as gossip protocols, were introduced in the context of replicated database consistency management [3]. They were first used to implement reliable broadcast in large networks in [2]. This latter protocol proceeds in two phases. In the first phase, processes use an unreliable gossip-based dissemination of information to transmit messages; in the second phase, messages losses are detected and repaired via re-transmissions. Many variations of this protocol have been proposed, mostly orthogonal to the ideas described in our paper. Improved buffering techniques, for example, have been considered in [6] and [9]. In both cases, the goal is to limit the amount buffering required for a message. While the former work requires a full knowledge about the system membership, the latter does not. The approach in [9] is mainly concerned with process recovery. Alternative approaches have considered recovering messages from the sender's log [12]. In [6], heuristics are presented to garbage collect messages. The approach aims to identify "aging" buffered messages.

The only adaptive gossip-based reliable broadcast protocol we are aware of is [11]. In this protocol, processes adjust the message rate emission to the amount of resources available (i.e., buffer size) and to the global level of congestion in the system. Processes periodically evaluate the available resources in the system and from time to time exchange the minimum buffer size. Senders then reduce their gossip rate according to their estimates about the mean number of messages in a process' buffer. We are not concerned with adjusting sending rates in this work, and the ideas described in this work could be easily integrated in our algorithm. Control information, for example, used in both algorithms could be combined into a single message.

In [4] and [5] the authors show how to implement a gossip-based reliable broadcast protocol in an environment in which processes have a partial view of the system membership. Our approach does not require processes to know all the system members or the topology connecting them. This information, however, allows processes to improve their gossiping.

Reducing the number of gossip messages exchanged between processes by taking the network topology into account is discussed in [7] and [8]. Processes communicate according to a predetermined graph with minimal connectivity to attain a desired level of reliability. Similarly to our approach, the idea is to define a directed spanning tree on the processes. Differently from ours, no process and link reliability guarantees are taken into account to build such trees.

7 Concluding Remarks

This paper was motivated by a simple observation: typical gossip algorithms need to retransmit more messages than adaptive algorithms to reach the same reliability probability. Based on this observation, we proposed a new approach for broadcasting messages with a given reliability probability. For this purpose, we defined the notions of *optimal* and *adaptive* probabilistic reliable broadcast algorithms. We then proposed an algorithm that converges toward optimality, by adapting its behavior to the distributed environment in which it executes. When provided with exact knowledge about failure probabilities, we proved that our adaptive algorithm is indeed optimal. We also evaluated the performance of our algorithm through simulation and showed that it quickly converges toward exact knowledge of failure probabilities.

We plan to pursue this work in several directions. First, we intend to apply our approach to distributed problems other than reliable broadcast and to consider optimality criteria different than the number of messages. Another idea is to improve our statistical inference mechanism, for example by dynamically increasing the number of probabilistic intervals when better precision is required. Along the simulation axis, we also plan to gather further results based on more complex topologies. For example, our current simulations rely on the conservative assumption that all failure probabilities are identical. By revisiting this assumption, we expect our adaptive algorithm to further increase its performance gain with respect to typical gossip algorithms.

References

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data structures and algorithms.* Addison Wesley, 1987.

[2] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, Canada, August 1987.

[4] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 443–452, July 2001.

[5] A.-M Kermarrec, L. Massoulie, and A.J. Ganesh. Probabilistic reliable dissemination in large-scale systems. Technical report, Microsoft Research, June 2001.

[6] P. Kouznetsov, R. Guerraoui, S.B. Handurukande, and A.-M. Kermarrec. Reducing noise in gossip-based reliable broadcast. In *Proceedings of the 20th International Symposium on Reliable Distributed Systems*, pages 186–189, New Orleans, LA, USA, October 2001.

[7] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. Technical Report CS1999-0622, University of California, San Diego, June 1999.

[8] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministic flooding: Low message overhead and high reliability for broadcasting on small networks. Technical Report CS1999-0637, University of California, San Diego, November 1999.

[9] O. Ozkasap, R. van Renesse, K. Birman, and Z. Xiao. Efficient buffering in reliable multicast protocols. In November, editor, *Proceedings of International Networked Group Communication*, Pisa, Italy, 1999.

[10] J. Pereira, L. Rodrigues, M. J. Monteiro, R. Oliveira, and A.-M. Kermarrec. Neem: Network-friendly epidemic multicast. In *Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, Florence,Italy, October 2003.

[11] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 47–56, San Francisco (CA), USA, June 2003.

[12] Q. Sun and D. Sturman. A gossip-based reliable multicast for large-scale high-throughput applications. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, New York (USA), June 2000.

A Typical vs. Adaptive Gossip Algorithm

The example presented in Section 1 consists of nodes N_1 and N_2 connected through two independent paths. Path one has a loss probability of L; path two has a loss probability of αL, where $\alpha > 1$. After k_0 messages are transmitted with a typical gossip algorithm, $k_0/2$ go through path one and $k_0/2$ through path two. Thus, the probability that a message reaches N_2 through path one and two is, respectively, $1 - L^{k_0/2}$ and $1 - (\alpha L)^{k_0/2}$. And the probability that a message reaches N_2 through any path is $1 - L^{k_0/2}(\alpha L)^{k_0/2} = 1 - (\sqrt{\alpha}\, L)^{k_0}$. With an adaptive algorithm, all messages go through path two, which has a smaller loss probability. The probability that at least one out of k_1 messages reaches N_2 is then $1 - L^{k_1}$. We can then determine the relation k_1/k_0 when both methods lead to the same probability: $1 - (\sqrt{\alpha}\, L)^{k_0} = 1 - L^{k_1}$, which can be developed into $k_1\ log\, L = k_0\ log\,(\sqrt{\alpha}\, L)$, resulting in $k_1/k_0 = 0.5\ log_L\, \alpha + 1$.

B Maximum Reliability Tree Algorithm

The Maximum Reliability Tree is a spanning tree containing the most reliable paths in $G = (\Pi, \Lambda)$ connecting all processes in Π. Algorithm 6 implements function $mrt(G, C)$ using a modified version of Prim's algorithm [1]. Function $mrt(G, C)$ returns graph (Π, Γ), where $\Gamma \subseteq \Lambda$ contains exactly $|\Pi| - 1$ links. C is used in $mrt()$ to determine the reliability of the links.

Algorithm 6 Maximum Reliability Tree (MRT)

```
1: function $mrt(G, C)$
2:     $\Gamma \leftarrow \emptyset$
3:     $S \leftarrow \{p_1\}$
4:     while $\Pi \setminus S \neq \emptyset$ do
5:         $R = \{l_{u,v} \mid l_{u,v} \in \Lambda$ and $p_u \in S$ and $p_v \in \Pi \setminus S\}$
6:         let $l_{u,v} \in R$ such for all $l_{r,s} \in R$: $(1-P_u)\times(1-L_{u,v})\times(1-P_v) \geq (1-P_r)\times(1-L_{r,s})\times(1-P_s)$
7:         $\Gamma \leftarrow \Gamma \cup \{l_{u,v}\}$
8:         $S \leftarrow S \cup \{p_v\}$
9:     return $(\Pi, \Gamma)$
```

C Optimality Proof of MRT

We initially prove the following lemmata. A link in a graph between processes p and q means that either p sends a message to q, or q sends a message to p, or both.

Lemma 1 *Every optimal algorithm propagates messages according to some tree $T \subseteq G$.*

Proof (sketch): The proof is by contradiction. Assume T is not a tree. Then there is necessarily a cycle in T. Let $R \subseteq T$ such that R is a cycle with the minimum number of processes and links. Let p be a process in R that receives a message from a process not in T, or p is the source. Since T is optimal, p clearly does not receive any messages from its left and right neighbors. Thus, p sends messages to its both neighbors. Without loss of generality, assume q is p's right neighbor. So, q does not send a message to p but to its other neighbor. Applying a recursive reasoning, and from the fact that R is a cycle, we conclude that p's left neighbor sends a message to p. A contradiction. □

Lemma 2 *MRT propagates fewer messages than any other spanning tree in G.*

Proof (sketch): Let U be an undirected graph that represents the network. An edge connects nodes a and b in U if and only if a and b are neighbors in the original network, and its value is given by $1-(1-P_a)\times(1-L_{a,b})\times(1-P_b)$. Hence, the MRT is exactly the maximum spanning tree of U and one important propriety of it is that it is always possible to find a bijection B between its edges and the edges of any other spanning tree ST, such that every edge in the maximum spanning tree has a higher or equal value than its relative edge in ST. This means that a different tree will have to send at least the same amount of messages through its channels in order to have a message reliably broadcast. □

Theorem 1 *Our MRT-based algorithm is optimal.*

Proof: Immediate from Lemmata 1 and 2. □

D Optimality Proof of *optimize*()

In order to simplify the optimality proof of our greedy algorithm, we first operate a problem transformation by swapping the objective function $c(\vec{m})$ and the constraint function $r(\vec{m})$ in Eq. (3). Such a transformation results in the optimization problem presented in Eq. (5).

$$\begin{array}{lrcll} \textit{maximize} & r(\vec{m}) & = & \prod_{j=1}^{|\vec{m}|} 1-\lambda_j^{m[j]} & \\ \textit{subject to} & c(\vec{m}) & = & \sum_{j=1}^{|\vec{m}|} m[j] & \leq \quad M \end{array} \tag{5}$$

In this new problem, M is a number of messages given *a priori* that constraints function $c(\vec{m})$, whereas the probability to reach all processes represented by function $r(\vec{m})$ has now to be maximized. It is straightforward to see that the two problems are equivalent, i.e., any solution to one of these problems can be derived into a solution to the other problem.[3] For conciseness sake, we do not prove the corresponding Lemma 3.

Lemma 3 *The two optimization problems of Eq. (3) and Eq. (5) are equivalent.*

For our proof, we also need to express the gain function $\alpha()$ in a way that allow us to reason about it properties; this is the purpose of Eq. (6). From this expression, a straightforward induction proof shows that $\alpha()$ is *isotonic*, i.e., non-decreasing. This property is illustrated by Eq. (7). Again, for conciseness sake, we do not prove the corresponding Lemma 4.

$$\alpha_r(\vec{m},\vec{m}_k) = \frac{1-\lambda_k^{m_k+1}}{1-\lambda_k^{m_k}} \tag{6}$$

$$\alpha_r(\vec{m},\vec{m}_k) = \frac{1-\lambda_k^{m_k+1}}{1-\lambda_k^{m_k}} \geq \frac{1-\lambda_k^{m_k+2}}{1-\lambda_k^{m_k+1}} = \alpha_r(\vec{m}+\vec{m}_k,\vec{m}_k) \tag{7}$$

Lemma 4 *The gain function $\alpha()$ is isotonic.*

Given Lemmata 3 and 4, we can now prove that Algorithm 2 solves the optimization problem of Eq. (3), by showing that Eq. (5) defines an equivalent *greedy problem.* This is the aim of Lemma 5 and Theorem 2 presented hereafter.

[3]Furthermore, if we replace the stop condition in Algorithm 2 with $c(\vec{m}) = M$, we immediately get a greedy algorithm for the new problem.

Lemma 5 *Eq. (5) defines a greedy problem.*

PROOF: We have to prove that the problem defined by Eq. (5) exhibits the two classical properties known as the *Greedy Choice Property* and the *Optimal Substructure Property.*

Greedy Choice Property. We have to show that the first step taken by our greedy algorithm, say $g = \vec{u}_k$, can be part of *some* optimal solution. Let $\vec{m}_s$ be an optimal solution that does *not* contain step g. From this, we know that $\vec{m}_s = \sum_i m_s[i] \times \vec{u}_i$ with $m_s[k] = 1$. Then, from Lemma 4, we know that if the gain is maximum for $\vec{u}_k$ on the first step of Algorithm 2, i.e., when $\forall i : \; m_s[i] = 1$, this is still the case if $\exists i \neq k$ such that $m_s[i] > 1$. From this, we conclude that the solution $\vec{m}_g = \vec{m}_s - \vec{u}_i + \vec{u}_k$, which contains our first greedy step g, is as good as solution $\vec{m}_s$. Hence $\vec{m}_g$ is an optimal solution as well.

Optimal Substructure Property. From the above, we know that there exists an optimal solution $\vec{m}_g$ that includes our first greedy step $g = \vec{u}_k$. We now have to show that after that first step, we are left with a subproblem which is optimally solved in $\vec{m}'_g = \vec{m}_g - \vec{u}_k$. Let P be the original problem given by Eq. (5) and P' the subproblem obtained after applying step g. Eq. (8) below expresses the new problem P'.

$$\begin{array}{lllcccccl} \textit{maximize} & r'(\vec{m}) & = & \prod_{j=1}^{k-1} 1 - \lambda_j^{m[j]} & \times & (1 - \lambda_k^{m[k]+1}) & \times & \prod_{j=k+1}^{|\vec{m}|} 1 - \lambda_j^{m[j]} & \\ \textit{subject to} & c'(\vec{m}) & = & \sum_{j=1}^{k-1} m[j] & + & (m[k]+1) & + & \sum_{j=k+1}^{|\vec{m}|} m[j] & \leq \; M \end{array} \tag{8}$$

We show that $\vec{m}'_g$ is an optimal solution for P' *by contradiction.* Assume that there exists a solution to P', say $\vec{m}_s$, such that $c'(\vec{m}_s) \leq M$ and $r'(\vec{m}_s) > r'(\vec{m}'_g)$. On the other hand, we have that $r'(\vec{m}'_g) \; = \; r(\vec{m}_g)$ is maximum under the constraint $c(\vec{m}_g) = c'(\vec{m}'_g) \leq M$. A contradiction. $\square$

Theorem 2 *Algorithm 2 solves the optimization problem of Eq. (3).*

PROOF: Immediate from Lemmata 3 and 5. $\square$